Qualitätssicherungskonzept

Allgemein

Es wird das Extreme Programming Paradigma mit pair programming (PP) verwendet. Dabei wird das Produkt sinnvoll in kleinere Einheiten die sog. Stories zerlegt, welche dann parallel von den Paaren implementiert werden. Bei Aufteilung der Paare ist auf aufgeglichene programmierfertigkeiten zu achten, so dass sich keine "Expertenteams" bilden, sondern der Quellcode allen gehört und auch von allen verändert werden kann.

Dem folgend ist von jedem PP zeitgleich zur Implementierung einer User-Story die betreffende Dokumentation zu erstellen (Quelltextdokumentation, Benutzerhandbuch). Dem Prinzip "Tests first" folgend werden vor Implementierugn einer User-Story entsprechende Tests erstellt. Jedes PP testet nach Fertigstellung der Komponeneten mit den betreffenden Tests diese selbstständig und ergänzt die Testdokumentation entsprechend. Die einzelnen Komponenten werden in wöchentlichen Integrationsphasen zu einem neueren Release zusammengefügt, in dessen Rahmen die Integrationstests erfolgen.

Dokumentationskonzept

Quelltextorientierte Dokumentation

Zum leichteren Verständnis des Codes ist eine Kommentierung unerlässlich.
Für Beschreibungen von Klassen, Klassenvariablen oder Methoden werden Dokumentationskommentare gesetzt, dabei soll eine Beschreibung erläutern, welche Funktionen die Variable, Methode bzw. Klasse erfüllt und nicht die konkrete Umsetzung. Einzuleiten sind solche Kommentare so: /**comments*/, wobei jede Zeile des Kommentars mit „*“ beginnt. Aus solch einer Kommentierung lässt sich später eine javadoc-Dokumentation generieren.
Kommentare von geringem Umfang, die nicht der Beschreibung dienen, sondern ausschließlich dem leichterem Verstehen der Implementierung, sind am Ende der betreffenden Zeile zu setzen.
z.B. `difficultassigment; //explaination`
Dennoch soll nicht für jede Anweisung ein Kommentar geschrieben werden, sondern eher überlegt werden, ob der Code nicht verbesserungswürdig ist.
Implementierungsbeschreibung größeren Umfangs werden in Block-Comments geschrieben /*.....*/,wobei jede Zeile mit einem „*“ beginnt.

Benutzerdokumentation

Zeitgleich mit der Implementierung wird ein Benutzerhandbuch erstellt, das den Umgang mit der Applikation und alle Funktionen, die das Programm bietet, beschreibt und Systemvoraussetzungen, die zum Benutzen des Programms erforderlich sind, genannt. Dieses wird nach Fertigstellung einer User-Story von den entsprechenden PP ergänzt.

Entwurfsdokumentation

In der Entwurfsdokumention wird das Designkonzept des Programms beschrieben. Untergliedert wird das Designkonzept in High-Level-Design und Low-Level-Design. Beim High-Level-Design wird eine Übersicht aller Pakete, das Zusammenspiel dieser und wie sie sich in das prefuse-Konzept eingliedern, dargelegt. Beim Low-Level-Design wird vertieft auf die Funktionsweise der einzelnen Module eingegangen, mithilfe der UML oder einer Beschreibung wichtiger Algorithmen oder Funktionsmechanismen.

Makrostrukturierung

Ordnerstruktur

Jede java-Datei ist nach dem prefuse- Architekturkonzept zu speichern, das heißt Komponenten, die auf die selbe Funktionalität abzielen z.B. DataTables (umfasst alle Klassen die zum Auslesen der Daten sowie das Speichern in prefuse spezifische Speicherstrukturen erforderlich sind) sind im selben Ordner zu speichern, der für die jeweilige prefuse-Schicht vorgesehen ist. Dadurch lässt sich eine strikte Einhaltung des prefuse-Konzeptes realisieren.

Externe Dokumentierung der Ordner

Jeder Ordner beinhaltet eine README.txt-Datei, die das Zusammenspiel der Klassen innerhalb des Paketes beschreibt und das resultierende Ergebnis, das durch dieses Paket erzielt wird.

Namenskonvention der Ordner

Ordner sind nach den Schichten von prefuse zu bezeichnen, um Fremdentwicklern eine leichtere Orientierung an dem IVRM-Modell zu gewährleisten. Ausnahmen sind Pakete, die unabhängig vom Prefuse-Konzept Funktionalitäten beinhalten. Diese sind mit sprechenden Namen zu bezeichnen, wobei zusammengesetzte Wörter erlaubt sind.

Codestruktur

Klassen/Interface-Aufbau

- Klassenkommentierung(Spezifikation s. Kommentierung)
- Klassen/Interface-name
- klassenstatische Variable (public, protected, private)
- Instanzvariablen(public, protected. Private)
- Kontruktor (bei mehreren Kontruktoren erst die allgemeineren, dann die speziellen)
- Methoden

Namenskonventionen

Alle Klassen, Attribute und Methoden sind mit englischen Vokabular zu bezeichnen. Des Weiteren ist darauf zu achten, dass „sprechende Namen“ vergeben werden. Bei zusammengesetzten Wörtern soll das zweite Wort wieder mit einem Großbuchstaben beginnen.

- Klassennamen sind mit Großbuchstaben zu beginnen und bestehen aus einem Substantiv.
- Objektnamen beginnen mit einem Kleinbuchstaben und werden nach ihrer Klasse bezeichnet. Bei Interaktionselementen einer GUI fangen Objektnamen mit dem Attributnamen der Fachkonzeptklasse an, gefolgt vom Namen des Interaktionselements.
- Attributnamen sollen den funktionalen Sinn des Attributes aufzeigen.
- Operationsnamen beginnen in der Regel mit einem Verb gefolgt von einem Substantiv. Lesende und schreibende Methoden erfolgen durch get- und set- Methoden. Die strukturell so aufgebaut sind: getAttributname bzw. setAttributname .

Deklarierung von Variablen

Jede Deklaration einer Variable wird in einer extra Zeile vorgenommen. Variablen, die in Blöcken deklariert werden sind am Anfang eines Blockes zu deklarieren. Ausnahmen sind Variablendeklarationen bei for-Anweisungen.

Einrückungen, Leerzeichen und Zeilenumbrüche

- Öffnende geschweifte Klammern stehen hinter der Deklaration einer Methode, einer Kontrollanweisung und einer Schleifendeklaration oder sonstigen Anweisungsblockdeklarationen in der selben Zeile. Schließende geschweifte Klammern eines Blockes stehen in einer extra Zeile. Code des dazugehörigen Blockes ist 4 Leerzeichen weiter als die Klammern ein zu rücken. Unterblöcke eines Blocks sind eine Einrückungsstufe weiter ein zu rücken, um eine gute Strukturierung zu gewährleisten
- Codezeilen sollten nicht länger als 70 Zeichen sein mit Ausnahme bei entstehender schwerer Lesbarkeit. In der Regel ist ein Zeilenumbruch vorzunehmen.
- Parameter von Methoden sind nach dem Komma mit Leerzeichen zu trennen, um ein einheitliches Bild zu sichern.

Organisatorische Festlegungen

Jedes Gruppenmitglied soll bestrebt sein, die oben genannten Richtlinien einzuhalten.
Für die Einhaltung der Richtlinien der Implementierung ist der Verantwortliche für die Implementierung zuständig sowie für die richtige Kommentierung des Codes. Diese Kontrolle erfolgt in kurzen Zeitintervallen, um Fehler schnell ausschließen zu können und eine schnellere Einarbeitung zu einem späteren Zeitpunkt durch die richtige Kommentierung des Codes zu gewährleisten. Bei Nichteinhaltung wird die Person, die für den Codeteil verantwortlich ist, kontaktiert, um den Fehler selber zu beheben.
Der Verantwortliche für die Dokumentation ist für die Erstellung der externen Dokumentation verantwortlich, das heißt er erstellt die Entwurfsdokumentation und das Benutzerhandbuch.
Jede Implementierung kann Fehler beinhalten, die nicht sofort ersichtlich sind, deshalb ist der Verantwortliche für Tests für die Korrektheit von Systemkomponenten zuständig. Entscheidende Komponenten sollen, wenn möglich, sofort getestet werden. Zur Kontrolle der Korrektheit werden vom Testverantwortlichen Testdokumente angefertigt, die bestandene Test darlegen.

Testkonzept

Zur Sicherung der Qualität muss im Laufe des Praktikums eine geeignete Serie von Tests ausgearbeitet werden, die jedem Teammitglied das programmierbegleitende Testen von einfacheren bis zu komplexen Zusammenhängen ermöglicht. Die Idee hinter dem Testen ist die, dass man eine Möglichkeit hat neben den syntaktischen Fehlern auch die semantischen zu erkennen. Im Vordergrund der von uns zu entwickelnden Applikation stehen Zuverlässigkeit, Funktionalität und Effizienz. Der Benutzer unserer Applikation kann potentiell mit großen Datensätzen zu tun haben. Daher ist besonderes Augenmerk auf das Testen möglicher Zeitverzögerungen zu legen. Ein rechtzeitiges Testen wird uns die Möglichkeit geben, Fehler sowie unzureichendes Zeitverhalten früh zu erkennen und zu beheben.

Bei der Entwicklung unseres Projekts wird zu Testzwecken das Werkzeug JUnit verwendet.

Die Tests des Projektes werden in 4 Phasen gegliedert:

1. Komponententest
2. Integrationstest
3. Systemtest
4. Abnahmetest

JUnit Framework

JUnit ist ein Test-Werkzeug für Java. Es ist für automatisierte und gezielte Tests von funktionalen Einheiten (meistens Klassen und Methoden) auf ihre Korrektheit besonders geeignet. Dabei ist Methode die kleinste Einheit, die mittels JUnit geprüft werden kann. Die Testklassen werden als Unterklassen von `junit.framework.TestCase` angesehen und können auf diese Weise die Methoden der Oberklasse aufrufen. Die wichtigste Funktionalität von JUnit bereiten die Methoden der Klasse `Assert`. Darunter:

`assertEquals(Object expected, Object actual)` Prüfung auf Gleichheit von Objekten
`assertFalse(boolean condition)` Fehler, wenn boolescher Ausdruck wahr
`assertNull(Object object)` Prüfung ob `object` gleich "null" ist

Ein Vorteil des Werkzeugs ist, dass der eigentliche Code der Applikation separat von dem Testcode gehalten wird, insbesondere in einem eigenen Paket.

1. Der Komponententest

Komponententests werden verwendet, um einzelne Klassen und Methoden auf Korrektheit zu überprüfen. Zu den zu testenden Klassen werden Testklassen mit Testmethoden erzeugt und mit Funktionalität der JUnit-Klassen (hauptsächlich junit.framework.Assert) ausgestattet. Testklassen können im vorgegebenem Umfang zu Testsuiten zusammen gefasst werden. Beim Aufruf einer Suite werden dann alle darin enthaltenen Tests gleichzeitig durchgeführt. Das TestRunner-Werkzeug, welches Teil der JUnit-Umgebung ist, führt die Tests aus und berichtet die Ergebnisse. JUnit kennt dabei 2 Ergebnisarten: entweder ist ein Test gelungen oder misslungen. Sollte ein Test nicht gelingen, wird dies auf zweierlei Weise angedeutet: als Fehler (Error) bzw. als falsches Ergebnis (Failure). Failure wird üblicherweise mit der Exception *AssertionFailedError* angezeigt.

Zu verwendete Syntaxkonvention:
Suffix *Test* bei Testklassen: *Klassename**Test***
Suffix *Testsuite* bei Testsuiten: *Paketname**Testsuite***

2. Der Integrationstest

Die einzelnen Komponenten haben erfolgreich die Komponententests bestanden. Die Funktion des Integtationstests besteht darin, die verschiedenen voneinander abhängigen Komponenten des Systems und ihre Schnittstellen zusammen auf Interaktion zu überprüfen. Dabei sollen die interagierenden Module möglichst ausschöpfend bestimmt und entsprechend den Tests unterzogen werden. Für unser Projekt sind dabei von Bedeutung u.a. Einlesen und Transformation von Quelldaten sowie Abbildung der Daten auf Graphen.

3. Der Systemtest

Das Ziel des Systemtestes ist die Prüfung des Softwareprodukts. Dabei müssen alle im Pflichtenheft erfassten Funktionalitäten im Bezug auf ihre Implementierung getestet werden. Der Test wird aus Sicht des künftigen Anwenders durchgeführt, d.h. auf Ebene der Benutzeroberfläche. Im Vordergrund stehen besonders Funktions-, Sicherheits- sowie Interoperabilitätstests. Wesentlich in dieser Phase ist aussagekräftige Testfälle zu finden und mit ihnen zu arbeiten, weil eine Automatisierung des Vorgangs bei Systemprüfung in der Regel nicht möglich ist und nicht alle möglichen Szenarios berücksichtigt werden können.

Anschließend werden nicht-funktionale Anforderungen getestet, wie:

- Vollständigkeit
- Leistung
- Zuverlässigkeit
- Benutzbarkeit
- Robustheit

Dabei können bestimmte Module sowohl von unmittelbaren Entwicklern als auch von anderen Teammitgliedern geprüft werden. Testen durch am Projekt nicht beteiligte Personen kann ein Feedback bezüglich der Softwareergonomie, insbesondere Verständlichkeitsgrades der Funktionalität und Handhabbarkeit, liefern.

4. Der Abnahmetest

In der Abnahmephase wird dem Auftraggeber das Gesamtprodukt vorgestellt. Der Auftraggeber hat damit die Möglichkeit zu prüfen, ob das entwickelte Softwareprodukt die vereinbarten Anforderungen erfüllt.